Nondeterminism

- Nondeterminism is a theoretical idea, in general it means "don't make decision, consider all possible outcomes at the same time with same probability." Note that, it doesn't mean "randomly pick possible outcome". Just like Schrodinger' cat, it is both alive and dead at the same time with a 50% 50% probability.
  - For example, when we say, "choose one side of a coin nondeterministically", it means "choose head with 50% probability and choose tail with 50% probability." It doesn't "pick one side randomly" because it will result in either a head or a tail.
    - However, a nondeterministic program can be simulated by "random picking a branch among all possible branches for a lot of times". For example, you can simulate the procedure "choosing one side of a coin nondeterministically" by tossing a fair coin a lot of times, then you will come up with a set of outputs:$\{head,\ tail\}$, and the probability of two outputs will be near 50% and 50%.

  - Here is another example. If we have a simple program that returns the maximum between x and y:

    $$\textbf{if } x \le y \textbf{ then } max \coloneqq y \textbf{ else } max \coloneqq x \textbf{ fi}$$

    This $\textbf{if} - \textbf{else}$ statement is deterministic. When we have $x = y$, it will always choice y to be the max. Consider a different program that does the following:

    "If $x \ge y$ then $max \coloneqq x$, if $x \le y$ then $max \coloneqq y$"

    and it can evaluate both branches at the same time. When we have $x = y$, then max should have 50% chance to be x and 50% chance to be y.

- Actually, a machine/program cannot run *nondeterministically*, but designing a nondeterministic machine/program is useful:
  1. A nondeterministic machine/program has a deterministic machine/program that can do the same job (although the deterministic version might need to finish the same job using much longer time). This is a topic in CS530 Computational Theory.
  2. The design process can be simplified. We can avoid some insignificant choice-making and only focus on the important decisions in the design. The design of a nondeterministic machine is easier to read and understand.

(Syntax of Nondeterministic Statements in our Programming language)

- A nondeterministic $\textbf{if} - \textbf{fi}$ statement: $\textbf{if } B_1 \to S_1 \ \square\ B_2 \to S_2 \ \square \ldots \square\ B_n \to S_n \ \textbf{fi}$
  - We use box symbols to separate different arms.
  - Inside each arm we have a **guarded command** $B_i \to S_i$ which means "if **guard** B_i is true then it's okay to run S_i".
  - How it works:
    - If none of the guards B_i is true, it aborts with a **runtime error.**
    - If exactly one guard B_i is true, then it executes S_i.
    - If more than one guard is true, then ***choose one*** of the true arms ***nondeterministically*** and execute the corresponding S_i.
    - Here, "***choose one… nondeterministically***" means, the machine splits into several copies of itself, and each copy follows one of the possible arms in parallel; and each copy holds a probability so that the total probability adds up to 1.

2. Write a nondeterministic $\textbf{if} - \textbf{fi}$ statement that chooses the maximum between x and y.

$$\textbf{if } x \ge y \to max \coloneqq x \ \square\ x \le y \to max \coloneqq y \textbf{ fi}$$

3. Write a nondeterministic $\mathbf{if}-\mathbf{fi}$ statement to simulate $\mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{fi}$

$$\mathbf{if}\ B \to S_1 \mathbin{\square} \neg B \to S_2\ \mathbf{fi}$$

- We express a nondeterministic **while** loop using a nondeterministic $\mathbf{do}-\mathbf{od}$ statement: $\mathbf{do}\ B_1 \to S_1 \mathbin{\square} B_2 \to S_2 \mathbin{\square} \dots \mathbin{\square} B_n \to S_n\ \mathbf{do}$
  - How it works:
    - At the beginning of each iteration, check all guards.
    - If none of the guards B_i is true, then finish loop.
    - If exactly one B_i is true, then execute S_i and go to the next iteration.
    - If more than one guard is true, then choose one of the true arms *nondeterministically* and execute the corresponding S_i; then go to the next iteration.

(Denotational Semantics of Nondeterministic Program)

- Let $IF \equiv \mathbf{if}\ B_1 \to S_1 \mathbin{\square} B_2 \to S_2 \mathbin{\square} \dots \mathbin{\square} B_n \to S_n\ \mathbf{fi}$, $M(IF,\sigma) = \begin{cases} \{\bot_e\}, & \text{if } \sigma(B_1 \vee B_2 \vee \dots \vee B_n) = F \\ \{M(S_i,\sigma) \mid \sigma(B_i) = T\}, & \text{otherwise} \end{cases}$
- Let $DO \equiv \mathbf{do}\ B_1 \to S_1 \mathbin{\square} B_2 \to S_2 \mathbin{\square} \dots \mathbin{\square} B_n \to S_n\ \mathbf{od}$, $M(DO,\sigma) =$
$\begin{cases} \{\sigma\}, & \text{if } \sigma(B_1 \vee B_2 \vee \dots \vee B_n) = F \\ \{M(DO, M(S_i,\sigma)) \mid \sigma(B_i) = T\}, & \text{otherwise} \end{cases}$

- Also, in a nondeterministic program (usually in the loop) we might execute a statement S in a collection of (possibly pseudo) states. Let $\Sigma_0 \subseteq \Sigma_\bot$, then $M(S,\Sigma_0) = \{M(S,\sigma) \mid \sigma \in \Sigma_0\}$. Remind that, we usually use Σ to denote the set of all (well-formed) states. We denote $\Sigma_\bot = \Sigma \cup \{\bot\} = \Sigma \cup \{\bot_d, \bot_e\}$ here.

- Because of the nature of nondeterminism, we can end up with more than one state when we finish the execution of a nondeterministic program.
  - The denotational semantics of a deterministic program is one single (possibly pseudo) state: $M(S,\sigma) = \{\tau\}$ where $\langle S,\sigma\rangle \to^* \langle E,\tau\rangle$ and $\tau \in \Sigma_\bot$.
  - The denotational semantics of a nondeterministic program is a set of (possibly pseudo) states: $M(S,\sigma) = \{\tau \in \Sigma_\bot \mid \langle S,\sigma\rangle \to^* \langle E,\tau\rangle\}$.

4. Calculate denotational semantics for each of the following nondeterministic statements and states.
   a. $S \equiv \mathbf{if}\ T \to x \coloneqq 0 \mathbin{\square} T \to x \coloneqq 1\ \mathbf{fi}$, $\sigma = \emptyset$
   Since it is possible to have $\langle S,\sigma\rangle \to^* \langle E,\{x=0\}\rangle$ and $\langle S,\sigma\rangle \to^* \langle E,\{x=1\}\rangle$, thus $M(S,\sigma) = \{\{x=0\},\{x=1\}\}$.

   b. $S \equiv \mathbf{if}\ F \to x \coloneqq 0 \mathbin{\square} F \to x \coloneqq 1 \mathbin{\square} T \to \mathbf{skip}\ \mathbf{fi}$, $\sigma = \emptyset$
   There is only one true guard, so $\langle S,\sigma\rangle \to \langle \mathbf{skip},\sigma\rangle \to \langle E,\emptyset\rangle$, thus $M(S,\sigma) = \{\emptyset\}$.
   Remind that, we cannot omit the "{ }" when there is only $\emptyset$, "$M(S,\sigma) = \emptyset$" looks like we are saying S doesn't have denotational semantics.

   c. $S \equiv \mathbf{if}\ x \ge y \to max \coloneqq x \mathbin{\square} x \le y \to max \coloneqq y\ \mathbf{fi}$, $\sigma = \{x=1,\ y=1\}$
   No matter which arm, we have max gets bind with value 1. Thus $M(S,\sigma) = \{\{x=1, y=1, max=1\}\}$.

   d. $S \equiv \mathbf{do}\ x+y=2 \to x \coloneqq y/x \mathbin{\square} x+y=2 \to x \coloneqq x+1 \mathbin{\square} x+y=4 \to y \coloneqq x \mathbin{\square} x+y=4 \to x \coloneqq x-1;\ y \coloneqq y-1\ \mathbf{od}$, and $\sigma = \{x=1,\ y=3\}$
      - After the first iteration of the $\mathbf{do}-\mathbf{od}$ loop, we can have these states: $\{x=1, y=1\}$, $\{x=0, y=2\}$.

    - We execute the second loop in both above states. After the second iteration of the $\mathbf{do-od}$ loop, we can have these states: $\{x=1, y=1\}$, $\{x=2, y=1\}$, $\perp_e$ and $\{x=1, y=2\}$. We noticed that state $\{x=1, y=1\}$ appears again, and it is the only state that can pass some guard in the next iteration; if we keep evaluating S with $\{x=1, y=1\}$ the program will diverge.
    - Thus, $M(S,\sigma) = \{\{x=2, y=1\}, \{x=1, y=2\}, \perp_e, \perp_d\}$.

- From the above examples, we can see that:
  - The denotational semantics of a nondeterministic program can be only one state. Thus, we can say that "if $M(S,\sigma)$ is a set with more than one state, then S is nondeterministic" but its converse is not true.
  - For a nondeterministic program S, $M(S,\sigma)$ might contain pseudo states, and possibly more than one type of pseudo states.
  - We can say $\tau \in M(S,\sigma)$, it means that τ is one of the possible (pseudo) state after evaluating S in σ; there might be other states. Similarly, we can also say $\{\tau_1, \tau_2\} \subseteq M(S,\sigma)$, it means that τ_1, τ_2 are two of the possible (pseudo) states after evaluating S in σ.

5. Given three sorted (into non-decreasing order) $size-n$ arrays b_0, b_1 and b_2, are there valid indices k_0, k_1 and k_2 such that $b_0[k_0] = b_1[k_1] = b_2[k_2]$? Create a program in our language that can find a set of such k_0, k_1 and k_2 if they exist.

   - Here we use the most naïve idea: use k_0, k_1 and k_2 as pointers and scan three arrays until we find a solution or one of them reaches n.
   - If we use a deterministic program, I can expect that there will be many different cases in each iteration: we are looking at three values: $b_0[k_0]$, $b_1[k_1]$ and $b_2[k_2]$, each two of them can be $=, <$ or $>$.

   - It will be much easier if we only have two arrays, for example let's look at only $b_0[k_0]$ and $b_1[k_1]$:
     - If $b_0[k_0] = b_1[k_1]$, we are done.
     - If $b_0[k_0] < b_1[k_1]$, since arrays are sorted, increasing k_1 can only get a larger number in b_1, so we should increase k_0.
     - If $b_0[k_0] > b_1[k_1]$, like the previous case, we should increase k_1.

   - Thus, if we ignore b_2 and ignore the array size, we can come up with a partial solution to this question using a nondeterministic $\mathbf{do-od}$ statement immediately:
   $$\mathbf{do}\ b_0[k_0] < b_1[k_1] \to k_0 := k_0 + 1 \ \Box\ b_0[k_0] > b_1[k_1] \to k_1 := k_1 + 1\ \mathbf{od}$$

   - Similarly, for the other combinations of values, we can come up with some other partial solutions:
   $$\mathbf{do}\ b_0[k_0] < b_2[k_2] \to k_0 := k_0 + 1 \ \Box\ b_0[k_0] > b_2[k_2] \to k_2 := k_2 + 1\ \mathbf{od}$$
   $$\mathbf{do}\ b_1[k_1] < b_2[k_2] \to k_1 := k_1 + 1 \ \Box\ b_1[k_1] > b_2[k_2] \to k_2 := k_2 + 1\ \mathbf{od}$$

   - Then, we can see the beauty of using a nondeterministic program: we don't need to worry about the overlapping cases, and it is very easy to combine partial solutions. We can come up with the following program:

```
$KKK \equiv$
        $k_0 := 0; k_1 := 0; k_2 := 0;$
        $\mathbf{do}\ b_0[k_0] < b_1[k_1] \to k_0 := k_0 + 1$
          $\Box\ b_0[k_0] > b_1[k_1] \to k_1 := k_1 + 1$
          $\Box\ b_0[k_0] < b_2[k_2] \to k_0 := k_0 + 1$
          $\Box\ b_0[k_0] > b_2[k_2] \to k_2 := k_2 + 1$
          $\Box\ b_1[k_1] < b_2[k_2] \to k_1 := k_1 + 1$
        □ $b_1[k_1] > b_2[k_2] \to k_2 := k_2 + 1$ **od**
```

- Before discussing the possible outcomes, let's modify KKK. First, let's merge some arms, since each $k_i := k_i + 1$ appears twice. And we have:
  $KKK_1 \equiv$
  ```
      $k_0 := 0; k_1 := 0; k_2 := 0;$
      do $b_0[k_0] < b_1[k_1] \vee b_0[k_0] < b_2[k_2] \to k_0 := k_0 + 1$
       □ $b_0[k_0] > b_1[k_1] \vee b_1[k_1] < b_2[k_2] \to k_1 := k_1 + 1$
       □ $b_0[k_0] > b_2[k_2] \vee b_1[k_1] > b_2[k_2] \to k_2 := k_2 + 1$ od
  ```

- KKK_1 is already pretty good, but we can still simplify each guard. For example, in the first arm, we only need $b_0[k_0]$ less than something to increase k_0, so we can come up with:
  $KKK_2 \equiv$
  ```
      $k_0 := 0; k_1 := 0; k_2 := 0;$
      do $b_0[k_0] < b_1[k_1] \to k_0 := k_0 + 1$
       □ $b_1[k_1] < b_2[k_2] \to k_1 := k_1 + 1$
       □ $b_2[k_2] < b_0[k_0] \to k_2 := k_2 + 1$ od
  ```

  If all three guards are False, then we have $b_0[k_0] \geq b_1[k_1] \geq b_2[k_2] \geq b_0[k_0]$ which implies an equality among all three values.

  - Does $\perp_e \in M(KKK_2, \sigma)$?
    It is possible. We can increase the value some k_i from $n-1$ to n and have runtime error in the next iteration.

- Now, let's take care of $\perp_e$. We can simply add some bounds checks in each guard, then:
  $KKK_3 \equiv$
  ```
      $k_0 := 0; k_1 := 0; k_2 := 0;$
      do if $k_0 \geq n \wedge k_1 \geq n$ then $F$ else $b_0[k_0] < b_1[k_1]$ fi $\to k_0 := k_0 + 1$
       □ if $k_1 \geq n \wedge k_2 \geq n$ then $F$ else $b_1[k_1] < b_2[k_2]$ fi $\to k_1 := k_1 + 1$
       □ if $k_2 \geq n \wedge k_0 \geq n$ then $F$ else $b_2[k_2] < b_0[k_0]$ fi $\to k_2 := k_2 + 1$ od
  ```

- What can we get if we calculate $M(KKK_3, \sigma)$, where b_0, b_1 and b_2 are bound in the state σ?
  - If there is at least one set of k_0, k_1 and k_2 that satisfies the requirements, why is there a $\tau \in M(KKK_3, \sigma)$ with $\tau(b_0[k_0]) = \tau(b_1[k_1]) = \tau(b_2[k_2])$ (in other words, why can this program find such k_0, k_1 and k_2)?

    1, 3, 5
    1, 2, 5
    3, 5, 7

    In each iteration, nondeterministically, one of k_i will be increased by exactly 1. Thus, if the problem has a solution, k_i will not miss its correct index. Once k_i reaches that index, it will wait for the other two indices since $b_i[k_i]$ must has largest value at the stage.
    The loop will terminate with $\tau(b_0[k_0]) = \tau(b_1[k_1]) = \tau(b_2[k_2])$, otherwise at least one of the guards is true.

  - If there are multiple solutions, does the program find all of them?
    No. For each k_i, its value can increase by exactly 1 after each iteration, so to reach the set of "larger" solutions we must go through the set of "smaller" solutions. This is true for all i, so every k_i will reach the smaller solution first, and when they reach the smaller solution, the loop terminates.

  - Does $\perp_e \in M(KKK_3, \sigma)$?
    No.

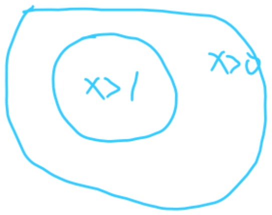
- For any $\tau \in M(KKK_3, \sigma)$, is it possible that $\exists 0 \le i \le 2. \tau(k_i) > n$?
  None of k_0, k_1 and k_2 can be bind with a value $> n$; because the value of some k_i can only increases after it passes the guard, and if the evaluation of $k_i \ge n$, then evaluating a guard will return $\perp_e$.

- Does $M(KKK_3, \sigma)$ contain more than one state?
  If there is at least one solution, then $M(KKK_3, \sigma) = \{\tau\}$ is only one state, and it contains the "smallest" solution. If there is no solution, then $M(KKK_3, \sigma)$ can be a set of states, where each state in the set contains some k_i (at least one) bound with value n.

Strength of Predicates

- A predicate can be considered as the collection of states that satisfy it; in other words, we can say $p = \{ \tau \in \Sigma \mid \tau \vDash p \}$.
  - For example, we can say $\{x = 1, y = 2\} \vDash x + y \le 4$ and we can also say $\{x = 1, y = 2\} \in x + y \le 4$.

- A predicate is **stronger** if it is "more restricted". For example, we can consider $x > 1$ is stronger than $x > 0$.
  - If we consider these predicates as collection of states, then $x > 1$ is a subcollection of $x > 0$.



- A predicate is **weaker** if it is "less restricted". For example, we can consider $x > 0$ is weaker than $x > 1$.
  - If we consider these predicates as collection of states, then $x > 0$ is a super-collection of $x > 1$.

- If predicates $p \Rightarrow q$, then p is **stronger** than q and q is **weaker** than p.
  - Technically, it should be "stronger than or equal to" and "weaker than or equal to". Because if $p \Leftrightarrow q$, then are equally strong.

6. What is the strongest predicate and what is the weakest predicate?
   - A predicate is weaker when it is less restricted, and more states can satisfy it. There is a predicate that all states satisfy it, which is T; so T is the weakest predicate.
   - On the hand, F is so restricted such that no state can satisfy it; so F is the strongest predicate.